Metoda Greedy

Metoda rezolvă probleme de optim în care soluția se construieşte pe parcurs. Optimul global se construieşte prin estimări succesve ale optimului local. Dintr-o mulțime A trebuie determinată o summulțime B, care verifică anumite **condiții** şi care de obicei este soluția unei probleme de **optimizare**.

Inițial mulțimea **B este mulțimea vidă**, se adaugă în B succesiv elemente, asigurându-se de fiecare dată un optim local, dar această construire nu asigură atingerea optimului global. De aceea Greedy nu poate fi aplicată decât dacă se demonstrează că modul de construire a lui B duce la obținerea unui optim global. De aceea Greedy se mai numeşte şi metoda optimului local.

Dacă în cazul unei probleme această metodă găseşte optimul pentru anumite date de test, dar pentru alte date nu găseşte soluția sau oferă o soluție care nu este optimă, spunem că metoda este euristică (în acest caz soluția poate fi determinată folosind tehnica backtracking, care presupune un algoritm de complexitate exponențială sau la unele probleme folosind programarea dinamică, acesta fiind cazul fericit).

Algoritmul general pentru Greedy

Cazul I

```
B = multimea vida
for (i=0; i<n; i++)
{
        x = alege( A);
        if (posibil( B ,x))
        * adauga elementul x la multimea B;
}
```

Cazul II

```
B = multimea vida
prelucreaza(A, v)
for (i=0; i<n; i++)
{
        x = v[i];
        if (posibil( B ,x))
        * adauga elementul x la multimea B;
}
```

Exemplu de problemă pentru care Greedy ne conduce la soluția optimă

Problema spectacolelor

Managerul artistic al unui festival trebuie sa selecteze o multime cat mai ampla de spectacole ce pot fi jucate in singura sala pe care o are la dispozitie.Stiind ca i s-au propus n spectacole si pentru fiecare spectacol i-a fost anuntat intervalul in care se poate desfasura [Si, Fi] (Si reprezinta ora si minutul de inceput, iar Fi ora si minutul de final al spectacolului i), scrieti un program care sa permita spectatorilor vizionarea unui numar cat mai mare de spectacole.

Date de intrare

Pe prima linie a fisierului de intrare ***spectacole.in*** se afla numarul n, numarul de spectacole propus. Pe urmatoarele n linii se vor afla 4 valori, primele doua reprezentand ora si minutul inceperii spectacolului curent, iar ultimele doua reprezentand ora si minutul terminarii spectacolului.

Date de iesire

Fisierul de iesire ***spectacole.out*** contine o singura linie, pe aceasta vor fi scrise numerele de ordine ale spectacolelor care indeplinesc solutia problemei, printr-un spatiu.

Restrictii

- n <= 100

Exemplu

spectacole.in	spectacole.out
5 12 30 16 30 15 0 18 0 10 0 18 30 18 0 20 45 12 15 13 0	5 2 4

Vom sorta crescator spectacolele dupa ora de final. Vom selecta initial primul spectacol (cel care se termina cel mai devreme). In continuare vom selecta, la fiecare pasa, primul spectacol neselectat, care nu se suprapune peste cele deja selectate.

O implementare intuitiva a acestui algoritm va fi prezentata in continuare. Pentru sortat vom folosi metoda BubbleSort, care este indeajuns de buna pentru limitele impuse de problema.

```
#include <iostream>
#include <fstream>
using namespace std;

ifstream f("spectacole.in");
ofstream g("spectacole.out");

int n,inceput[100],sfarsit[100],nr[100];

void citeste()
{
    int ora,min,i;
    f>>n;
    for (i=0;i<n;++i)
    {
        nr[i]=i+1;
        f>>ora>>min;
        inceput[i]=ora*60+min;
        f>>ora>>min;
        sfarsit[i]=ora*60+min;
    }
    f.close();
}

void sorteaza()
{
    int aux,schimb,i;
    do
    {
        schimb=0;
        for (i=0;i<n-1;++i)
            if (sfarsit[nr[i]]>sfarsit[nr[i+1]])
            {
                aux=nr[i];
                nr[i]=nr[i+1];
                nr[i+1]=aux;
                schimb=1;
            }
    }
```

```
    while (schimb);
}

void rezolva()
{
    int ultim,i;
    for (ultim=0,i=1;i<n;++i)
        if (inceput[nr[i]]>=sfarsit[nr[ultim]])
        {
            g<<nr[i]+1<<" ";
            ultim=i;
        }
    g<<endl;
}

int main()
{
    citeste();
    sorteaza();
    rezolva();
    return 0;
}
```

Exemplu de problemă pentru care Greedy nu ne conduce la soluția optimă

Problema rucsacului

Se considera ca dispunem de un rucsac cu capacitatea M si de N obiecte, definite fiecare prin greutate si valoare, ce trebuie introduce in rucsac. Se cere o modalitate de a umple rucsacul cu obiecte , astfel incat valoarea totala sa fie maxima. Nu putem lua fragmente dint-un obiect.

Date de intrare

Pe prima linie a fisierului de intrare *rucsac.in* se gasesc doua numere, primul fiind N, iar al doilea M (cu specificatiile din enunt). Pe urmatoarele N linii se gasesc, despartite printr-un spatiu valoarea obiectului curent si greutatea acestuia.

Date de iesire

In fisierul de iesire *rucsac.out* vor fi specificate numarul de obiecte şi numărul de ordine al obiectului.

Exemplu

rucsac.in	rucsac.out
5 10 1 1 4 8 3 3 5 15	24 4 2 1

Descrierea soluției folosind Greedy

Vom reprezenta solutia problemei ca pe un vector x. Vom ordona obictele descrescator tinand cont de valoarea/greutate. Atata timp cat obiectele incap in rucsac, le vom adauga in intregime, putem întâlni una din următoarele situații:

- obiectele alese au o greutate toatală egală cu a rucsacului
- mai există loc în rucsac, dar nu mai încape nici un obiect

Este evident că soluția găsită nu este întotdeauna optimă.

Probleme propuse (Greedy asigură soluția optimă)

1. Meniuri

La inaugurarea unui restaurant sunt prezente mai multe persoane. Clienții îşi aleg din meniul pus la dispoziție câte o specialiate. Dar deocamdată restaurantul are angajat un singur bucătar, care pregăteşte mâncărurile una după alta, deci clienții nu pot fi serviți decât pe rând. Presupunând că bucătarul se apucă de gătit după ce au fot strânse toate comenzile, stabiliți în ce ordine trebuie pregătite specialitățile, astfel încât timpul mediu de aşteptare al clienților să fie minim.

Date de intrare

Prima linie a fişierului ***meniu.in*** conține un număr natural n, reprezentând numărul clienților. Următoarea linie conține n numere întregi, reprezentând timpul necesar pregătirii fiecărei comenzi, în ordine pentru cele n persoane.

Date de ieşire

Fişierul meniu.out va conține două linii. Pe prima linie se va afişa un număr real cu 2 zecimale, reprezentând timpul mediu de aşteptare, iar pe a doua linie se vor scrie n numere naturale, reprezentând numerele de ordine ale persoanelor din restaurant în ordinea în care au fost servite.

Restricții

- 1<= n<=1000
- 1<=t<=100
- dacă sunt mai multe soluții se va afişa una singură

meniu.in	meniu.out
5 30 40 20 25 60	86.00 3 4 1 2 5

2. Cutii

Al Bundy este în dificultate. Are n+1 cutii de pantofi şi n perechi de pantofi identificate prin valorile 1, 2, ..., n(n perechi sunt aşezate în n cutii şi o cutie este liberă). Dar din păcate, pantofii nu se aflǎ la locurile lor. Pînă să vină Gary(şeful lui), Bundy trebuie să potrivească pantofii în cutii. Aranjarea pantofilor trebuie făcută rapid, astfel încât să nu trezească bănuieli. Prin urmare dacă scoate o pereche de pantofi dintr-o cutie trebuie să o pună imediat în cutia liberă. Ajutați-l pe Al Bundy să aranjeze pantofii la locurile lor printr-un număr minim de mutări.

Date de intrare

Prima linie a fişerului de intrare ***cutii.in*** conține numărul de cutii n. Linia a doua conține n+1 numere naturale distincte, separate prin spațiu, reprezentând dispunerea inițială a pantofilor în cutiile numerotate de la 1 la n+1. Printre cele n+1 numere numai unul are valoarea 0 şi corespunde cutiei goale. Linia a treia conține n+1 numere naturale, reprezentând configurația finală cerută, în care valoarea 0 corespunde cutiei goale.

Date de ieşire

Fişierul de ieşire ***cutii.out*** va conține pe prima linie un nu măr întreg k, reprezentînd numărul minim de mutări.

Restricții

1<n<100

cutii.in	cutii.out
4 3 4 1 0 2 4 0 2 1 3	4

3. Subşiruri

Se dă un şir de n numere întregi. Să se descompună acest şir în număr minim de subşiruri strict crescătoare, astfel încât numerele lor de ordine(din şirul dat) să fie ordonate cresscător în subşirurile formate.

Date de intrare
Prima linie a fişierului subşir.in conține un număr natural n, reprezentând numărul de elemente din şir. Următoarea linie conține cele n numere întregi separate printr-n spațiu.
date de ieşire
Date de ieşire
Fişierul de ieşire subşir.out va conține pe prima linie un număr întreg k, reprezentând numărul minim de subşiruri care se pot forma. Pe următoarele k linii vor fi descrise cele k subşiruri care se formează. Un subşir va fi precizat prin numerele de ordine ale elementelor din şirul inițial, separate printr-un spațiu.
Restricții

- 1<=n<=10000
- elementele şirului sunt numere cuprinse în intervalul [0,40000]

subsir.in	subsir.out
10 2 3 1 6 8 3 7 9 5 7	3 1 2 4 5 8 3 6 7 9 10

Idei de rezolvare

2.4.1. Meniuri

În această problemă se cere stabilirea unui minim. Se ştie că problemele de optim se pot rezolva aplicând metoda *greedy*, dacă se poate arăta că, pe baza alegerii optimului local, metoda generează soluții optime. Deci, mai întâi trebuie să demonstrăm că acest lucru este posibil.

În concluzie, ştiind că $t_{k_1} \leq t_{k_2} \leq ..., \leq t_{k_n}$ sunt timpii necesari preparării specialităților, presupunem că dacă acestea se prepară în ordinea $k_1, k_2, ..., k_n$, vom avea un timp total de aşteptare minim. Evident $\{k_1, k_2, ..., k_n\} = \{1, 2, ..., n\}$.

Demonstrația o facem prin *reducere la absurd*. Presupunem că ordinea $k_1, k_2, ..., k_n$ nu asigură timpul total minim de aşteptare. Din această presupunere temporară rezultă că există o altă ordine de servire a clienților (diferită de ordinea $k_1, k_2, ..., k_n$), care conduce la un timp total de aşteptare mai mic. Prèsupunem că o astfel de ordine diferă de ordinea $k_1, k_2, ..., k_n$ în pozițiile i şi j.

Conform ordinii $k_1, ..., k_i, ..., k_j, ..., k_n$ avem timpul total de aşteptare:
$timptotal(k_1, ..., k_i, ..., k_j, ..., k_n) =$
$= nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_i} + ... + (n-j+1)t_{k_j} + ... + t_{k_n}.$

Conform presupunerii de mai înainte, pentru ordinea $k_1, ..., k_j, ..., k_i, ..., k_n$ avem timpul total de aşteptare:
$timp_{total}(k_1, ..., k_j, ..., k_i, ..., k_n) =$
$= nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_j} + ... + (n-j+1)t_{k_i} + ... + t_{k_n}$

Conform presupunerii de mai înainte, pentru ordinea $k_1, \ldots, k_j, \ldots, k_i, \ldots, k_n$ avem timpul total de așteptare:

$$timp_{total}(k_1, \ldots, k_j, \ldots, k_i, \ldots, k_n) =$$
$$= nt_{k_1} + (n-1)t_{k_2} + \ldots + (n-i+1)t_{k_j} + \ldots + (n-j+1)t_{k_i} + \ldots + t_{k_n}$$

care este mai mic decât

$$nt_{k_1} + (n-1)t_{k_2} + \ldots + (n-i+1)t_{k_i} + \ldots + (n-j+1)t_{k_j} + \ldots + t_{k_n} =$$
$$= timp_{total}(k_1, \ldots, k_i, \ldots, k_j, \ldots, k_n).$$

Eliminând termenii asemenea din partea stângă, respectiv dreaptă a operatorului relațional avem:

$$nt_{k_1} + (n-1)t_{k_2} + \ldots + (n-i+1)t_{k_j} + \ldots + (n-j+1)t_{k_i} + \ldots + t_{k_n} <$$
$$nt_{k_1} + (n-1)t_{k_2} + \ldots + (n-i+1)t_{k_i} + \ldots + (n-j+1)t_{k_j} + \ldots + t_{k_n}$$

$$(n-i+1)t_{k_j} + (n-j+1)t_{k_i} < (n-i+1)t_{k_i} + (n-j+1)t_{k_j}$$
$$(n-i+1-n+j-1)t_{k_j} < (n-i+1-n+j-1)t_{k_i}$$
$$(j-i)t_{k_j} < (j-i)t_{k_i}$$

Deci, împărțind această relație cu $j - i$ (valoare pozitivă, deoarece am pornit cu presupunerea că $i < j$), avem $t_{k_j} < t_{k_i}$, ceea ce înseamnă că avem un timp de așteptare mai mic, după unul mai mare în șirul ordonat crescător a timpilor de așteptare, constatare care evident este o contradicție cu ipotezele fixate.

Pentru a minimiza timpul mediu de așteptare ($timp_{mediu}$) va trebui să minimizăm timpul total de așteptare ($timp_{total}$) al persoanelor pentru a fi servite, deoarece $timp_{mediu} = timp_{total}/n$. O persoană va trebui să aștepte prepararea meniului ei și a tuturor persoanelor care vor fi servite în fața lui. Intuitiv, dacă o persoană care dorește un meniu sofisticat este servită înaintea uneia al cărei meniu se prepară mai repede, timpul total de așteptare al celor doi este mai mare decât dacă servirea s-ar fi făcut invers. De exemplu, pentru timpii 30 și 20, în prima situație $timp_{total} = 30 + (30 + 20)$, iar în a doua situație $timp_{total} = 20 + (20 + 30)$.

În concluzie, vom alege persoanele în ordinea crescătoare a timpilor de preparare a meniurilor, ceea ce necesită ca ele să fie ordonate crescător în funcție de acest criteriu și vom calcula timpul total de așteptare ca fiind suma timpilor de așteptare a fiecărei persoane.

În situația în care există mai multe durate egale de preparare, nu contează ordinea alegerii acestora.

2.4.6. Cutii

Fie mai întâi exemplul din enunț. Avem 4 perechi de pantofi și 5 cutii în care acestea sunt aranjate în felul următor:

```
3 4 1 0 2
```

Se cunoaște și aranjarea dorită:

```
4 0 2 1 3
```

Adică, inițial, în prima cutie se găsește perechea 3, dar acolo trebuie să fie perechea 4, în a doua cutie se găsește perechea 4, dar a doua cutie trebuie să fie goală etc.

Un șir posibil de mutări (dar care nu conduce la soluția optimă) ar fi:

Mutarea 1: perechea 3 se mută din cutia 1 în cutia 4;
Mutarea 2: perechea 4 se mută din cutia 2 în cutia 1;
Mutarea 3: perechea 1 se mută din cutia 3 în cutia 2;
Mutarea 4: perechea 2 se mută din cutia 5 în cutia 3;
Mutarea 5: perechea 3 se mută din cutia 4 în cutia 5;
Mutarea 6: perechea 1 se mută din cutia 2 în cutia 4.

Am aranjat pantofii în **6** mutări.

Dar putem proceda în felul următor:

Mutarea 1: perechea 1 se mută din cutia 3 în cutia 4 (a ajuns unde trebuie, acum cutia 3 este goală);
Mutarea 2: perechea 2 se mută din cutia 5 în cutia 3 (a ajuns unde trebuie, acum cutia 5 este goală);
Mutarea 3: perechea 3 se mută din cutia 1 în cutia 5 (a ajuns unde trebuie, acum cutia 1 este goală);
Mutarea 4: perechea 4 se mută din cutia 2 în cutia 1 (a ajuns unde trebuie, acum cutia 2 este goală, așa cum cere configurația dorită).

Astfel am aranjat pantofii în **4** mutări.

Analizând exemplul, ajungem la concluzia că pentru a aranja pantofii în cutii, trebuie să mutăm fiecare pereche *p* de pantofi la locul ei (dacă nu este acolo). Există două

Analizând exemplul, ajungem la concluzia că pentru a aranja pantofii în cutii, trebuie să mutăm fiecare pereche *p* de pantofi la locul ei (dacă nu este acolo). Există două
cazuri posibile:

A. Cutia în care trebuie pusă perechea *p* este goală;
B. Cutia perechii *p* este ocupată de altă pereche *q*.

În cazul **A** vom muta perechea *p* la locul ei, în cutia goală, fără a mai face alte mutări. În cazul **B** trebuie eliberată mai întâi cutia ocupată de *q* și doar pe urmă vom putea muta perechea *p* în cutia ei, acum eliberată.

Situația **A** este de preferat, deoarece în acest caz o pereche ajunge la locul ei printr-o singură mutare. În situația **B** sunt necesare două mutări pentru a duce o pereche de pantofi unde trebuie. *Din acest motiv vom identifica în rezolvarea problemei cazul* **A** *cât timp este posibil* și o vom rezolva. În celelalte cazuri (similare situației **B**) vom aranja câte o pereche în cutia potrivită prin două mutări, conform explicațiilor date.

În șirurile ci și cf se memorează configurațiile inițială și finală a perechilor de pantofi în cutii. În variabilele cigol și cfgol se memorează indicele cutiei goale în configurația inițială și respectiv finală.

```
Subalgoritm Număr_mutări(n,ci,cf):
  repetă
    schimb ← fals                                        { încă nu am schimbat nimic }
    cât timp cigol ≠ cfgol execută:                      { rezolvăm situația A }
      ci[cigol] ← cf[cigol]          { mutăm în cutia goală perechea potrivită }
      cigol ← Caută(cf[cigol])
      ci[cigol] ← 0                                      { se eliberează o nouă cutie }
      mutări ← mutări + 1                                { am făcut doar o mutare }
      schimb ← adevărat                                  { am efectuat o schimbare }
    sfârșit cât timp
                                                         { se caută o cutie nearanjată }
    cât timp (ci[i] = cf[i]) și (i ≤ n+1) execută:
      i ← i + 1
    sfârșit cât timp
    dacă i ≤ n atunci                                    { rezolvăm situația B }
                             { golim cutia nearanjată și aducem perechea potrivită }
      ci[cigol] ← ci[i]
      ci[i] ← cf[i]
      cigol ← Caută(ci[i])
      i ← i + 1
      mutări ← mutări + 2                                { am făcut două mutări }
      schimb ← adevărat                                  { am efectuat o schimbare }
    sfârșit cât timp
  până când nu schimb      { până când nu a mai fost necesară nici o schimbare }
sfârșit subalgoritm
```

Observații

1. În algoritm s-a folosit funcția Caută(p) care returnează numărul cutiei care conține perechea *p*.
2. Situația A apare dacă indicele cutiei goale în configurația inițială este diferit de indicele cutiei goale în configurația finală.

2.4.7. Subșiruri

Subșirurile se construiesc simultan, printr-o singură parcurgere a șirului dat. Pentru fiecare element x_i din șir se caută un subșir existent la care acesta se poate adăuga la sfârșit. Dacă nu se găsește un astfel de subșir, vom crea unul nou în care x_i se va introduce ca prim element.

Exemplu

Fie $n = 8$ şi şirul de numere 9, 3, 10, 1, 11, 5, 12, 8

Pasul	Observaţii	Şir 1	Şir 2	Şir 3
1.	9 va fi primul element din primul şir	**9**	-	-
2.	3 va fi primul element din al doilea şir	**9**	**3**	-
3. (*)	10 se adaugă la sfârşitul şirului 1	9, **10**	**3**	-
4.	1 va fi primul element din şirul 3	9, **10**	**3**	**1**
5. (*)	11 se adaugă la sfârşitul şirului 1	9, 10, **11**	**3**	**1**
6. (*)	5 se adaugă la sfârşitul şirului 2	9, 10, **11**	3, **5**	**1**
7. (*)	12 se adaugă la sfârşitul şirului 1	9,10,11, **12**	3, **5**	**1**
8. (*)	8 se adaugă la sfârşitul şirului 2	9,10,11, **12**	3, 5, **8**	**1**

Observaţii

1. La paşii marcaţi cu (*) au existat mai multe posibilităţi de alegere a şirului la care să se adauge elementul curent.
2. Se observă că prin acest mod de construire a subşirurilor, elementele maxime din fiecare subşir (cele care s-au adăugat ultima oară) formează la rândul lor un şir descrescător (la pasul 4: 10, 3, 1, dar şi la pasul 8: 12, 8, 1). Dacă nu ar fi aşa, ar însemna că la un moment dat nu am ales în mod corect subşirul în care se adaugă un element. De exemplu, în tabelul de mai sus, elementul 10 nu poate fi pus după 3, deoarece el trebuie „înghesuit" (conform cerinţelor problemei) lângă 9. Din acest motiv, vom căuta cel mai potrivit subşir pentru un element cu algoritmul de căutare binară în şirul „vârfurilor" fiecărui subşir construit până la momentul respectiv. Astfel vom găsi mai rapid subşirul potrivit pentru elementul curent.

În următorul algoritm folosim notaţiile:

n: numărul elementelor în şirul dat;
x: şirul dat;
nrs: numărul subşirurilor crescătoare;
s: `s[i]` =numărul de ordine al subşirului din care face parte `x[i]`;
nrs: numărul subşirurilor crescătoare;
vf: valorile elementelor maxime din fiecare subşir.

```
Subalgoritm ConstruieşteSubşiruri(n,nrs,x,s,vf):
  nrs ← 1
  s[nrs] ← 1                              { x[1] face parte din subşirul 1 }
```

```
vf[nrs] ← x[1]
pentru j=2,n execută:                { se caută subşirul potrivit pentru x[j] }
  k ← Caută(vf,1,nrs,x[j])  { se apelează algoritmul de căutare binară }
  { care va returna numărul de ordine al subşirului unde ar trebui să se afle x[j] }
                                        { sau 0 dacă un astfel de subşir nu s-a găsit }
  dacă k = 0 atunci                     { dacă nu am găsit un subşir potrivit }
    nrs ← nrs + 1                              { x[j] va forma un nou subşir }
    s[j] ← nrs                           { numărul de ordine al subşirului }
    vf[nrs] ← x[j]            { valoarea elementului maxim din subşirul nrs }
  altfel                                 { x[j] se va adăuga unui subşir existent }
    s[j] ← k                             { numărul de ordine al subşirului }
    vf[k] ← x[j]               { valoarea elementului maxim din subşirul k }
  sfârşit dacă
sfârşit pentru
sfârşit subalgoritm
```

Chiar dacă algoritmul căutării binare se cunoaşte, prezentăm modul în care se aplică în rezolvarea acestei probleme, deoarece stabilirea numărului de ordine al subşirului în care se va adăuga elementul curent din şirul dat se realizează în cadrul acestui subalgoritm. Presupunem că în parametrul `ce` s-a transmis valoarea elementului `x[j]` curent. Variabila `găsit` o folosim în scopul de a reţine faptul că s-a găsit sau nu `ce` în şirul vârfurilor.

```
Subalgoritm Caută(vf,s,d,ce):
  găsit ← fals
  cât timp (s ≤ d) şi nu găsit execută:
    m ← (s+d) div 2
    dacă vf[m] = ce atunci
      găsit ← adevărat
    altfel
      dacă vf[m] > ce atunci
        s ← m + 1
      altfel
        d ← m - 1
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
  dacă găsit atunci         { dacă l-am găsit pe x[j] în capătul unui subşir }
                                        { avansăm în şirul vârfurilor }
    cât timp (ce = vf[m]) şi (m ≤ nrs) execută:
      m ← m + 1                       { cât timp se succede aceeaşi valoare }
    sfârşit cât timp
```

```
    dacă m ≤ nrs atunci      { dacă ne-am oprit după vârful unui şir existent }
      Caută ← m                            { aici se va adăuga x[j] }
    altfel
      { dacă şi ultimul şir îl are în vârf pe x[j], trebuie să creăm un subşir nou }
      Caută ← 0
    sfârşit dacă
  sfârşit dacă
  dacă nu găsit atunci                     { dacă nu l-am găsit pe x[j] }
    dacă (s ≤ nrs) şi (vf[s] < ce) atunci
                                { acolo unde a eşuat căutarea, şirul vf }
      Caută ← s      { conţine o valoare mai mică, am găsit locul lui x[j] }
    altfel
      Caută ← 0          { în caz contrar trebuie să creăm un subşir nou }
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

Arhiva educațională campion

1. reactivi

2. album

3. matriosca